Problem Set 4
data structures and algorithms 2 - fall 2023

yl2nr

due november 3, 2023 at 11:59 pm

Collaboration Policy: You are encouraged to collaborate with up to 4 other students, but all work submitted must be your own *independently* written solution. List the computing ids of all of your collaborators in the `collabs` command at the top of the tex file. Do not share written notes, documents (including Google docs, Overleaf docs, discussion notes, PDFs), or code. Do not seek published or online solutions for any assignments. If you use any published or online resources (which may not include solutions) when completing this assignment, be sure to cite them. Do not submit a solution that you are unable to explain orally to a member of the course staff. Any solutions that share similar text/code will be considered in breach of this policy. Please refer to the syllabus for a complete description of the collaboration policy.

Collaborators: list your collaborators

Sources: list your sources

problem 1 *Dynamic Programming*

1. If a problem can be defined recursively but its subproblems do not overlap and are not repeated, then is dynamic programming a good design strategy for this problem? If not, is there another design strategy that might be better?

   Solution:

   No, as the subproblem is not repeating, then saving the results of subproblems is not helpful and does not reduce running time.

   Some potential strategies are Divide and Conquer and Greedy Algorithm. They could be better in this situation.

2. As part of our process for creating a dynamic programming solution, we searched for a good order for solving the subproblems. Briefly (and intuitively) describe the difference between a top-down and bottom-up approach.

   Solution:

   The top-down approach means we solve the problem in the normal order. Starting with the higher level, we break the general high-level problem into smaller sub-problems. Then we recursively break down problems into smaller sub-problems. When solving each sub-problem, we first check if we have solved it before according to the memory we cached. If not, then we solve it and store its result in the memory.

   The bottom-up approach means we start from the base-level smallest problems. Without recursion, we simply solve each subproblem and store its result for future reference when solving higher-level problems.

problem 2 *Birthday Prank*

Prof Hott's brother-in-law loves pranks, and in the past he's played the nested-present-boxes prank. I want to repeat this prank on his birthday this year by putting his tiny gift in a bunch of progressively larger boxes, so that when he opens the large box there's a smaller box inside, which contains a smaller box, etc., until he's finally gotten to the tiny gift inside. The problem is that I have a set of n boxes after our recent move and I need to find the best way to nest them inside of each other. Write a **dynamic programming** algorithm which, given a $fits(b_i, b_j)$ function that determines if box b_i fits inside box b_j, returns the maximum number of boxes I can nest (i.e. gives the count of the maximum number of boxes my brother-in-law must open).

Solution:

1. The idea is to top-down dynamic programming at any given box b_i and return the value $numofbox_i$. Then after finding the number of boxes nested for all box i and storing in $memo[i]$, we return the maximum value in *memo* as the final answer.

2. Let the function of recursion be $findNumberBox()$. Let the memory list be *memo*, default to be None in each entry. At each box b_k, we call $findNumberBox(b_k)$. In the function, we first check if $memo[k]$ has value. If $memo[k]$ is not None, we return $memo[k]$. Otherwise, we traverse all other boxes and return the subproblem with the maximum value of num of boxes returned. Let $numOfBox_k = 0$. For each b_i, where $i \neq k$, we first check $fits(b_k, b_i)$. If true, if $1 + findNumberBox(b_i) > numOfBox_k$, update $numOfBox_k = 1 + findNumberBox(b_i)$. If false, we do nothing. After traversing all other boxes, we update $memo[k] = numOfBox_k$. The base case is the smallest box and no boxes can fit in, so returns the value of 0.

3. Repeat step 2 for all boxes and stop when *memo* has no None value in it. Then we return the highest value in *memo*, which is the maximum number of boxes we can nest.

PROBLEM 3 *Arithmetic Optimization*

You are given an arithmetic expression containing n integers and the only operations are additions ($+$) and subtractions ($-$). There are no parenthesis in the expression. For example, the expression might be: $1+2-3-4-5+6$.

You can change the value of the expression by choosing the best order of operations:

$$((((1+2)-3)-4)-5)+6 = -3$$
$$(((1+2)-3)-4)-(5+6) = -15$$
$$((1+2)-((3-4)-5))+6 = 15$$

Give a **dynamic programming** algorithm that computes the maximum possible value of the expression. You may assume that the input consists of two arrays: `nums` which is the list of n integers and `ops` which is the list of operations (each entry in `ops` is either `'+'` or `'-'`), where `ops[0]` is the operation between `nums[0]` and `nums[1]`. *Hint: consider a similar strategy to our algorithm for matrix chaining.*

Solution:

The general idea is to apply top-down dynamic programming. Starting from the top-level problem, we backtrack to recursively calculate the maximized outcome by finding maximum result of different possible divisions of the top-level calculation into combinations of smaller subproblems.

1. Let the function be $maximizeResult()$. Let the memory matrix be *memo*. We start with the whole arrays, so call $maximizeResult(numarray = nums, opsarray = ops, start = 0, end = n)$. Assume the total number of numbers is n, then there are n different ways to split the array into two pieces and calculate the maximum as the maximum of the two parts. Thus, to find the maximum of such splits,

   $maximizeResult(nums, ops, start = 0, end = n)$

   $= \max maximizeResult(nums[start : k+1], ops[start : k], start = start, end = k), operatorops[k],$ $maximizeResult(nums[k+1 : end+1], ops[k : end+1], start = k+1, end = end)$, for $start \leq k \leq end$,

   where $operatorops[k]$ can be plus or minus according to $ops[k]$.

2. We continue to recursively split the each part of the array into smaller two parts, until we reach the base case of there is only one number in the array, and we return the number.

After calculation of each sub-problem, we store the value of
$maximizeResult(numarray, opsarray, start, end$ into $memo[start][end]$ for future reference in recursions.

3. Finally, we return the result of $maximizeResult(numarray = nums, opsarray = ops, start = 0, end = n)$, which equals to $memo[0][n]$

PROBLEM 4 *Stranger Things*

The town of Hawkins, Indiana is being overrun by interdimensional beings called Demogorgons. The Hawkins lab has developed a Demogorgon Defense Device (DDD) to help protect the town. The DDD continuously monitors the inter-dimensional ether to perfectly predict all future Demogorgon invasions.

The DDD allows Hawkins to predict that i days from now a_i Demogorgons will attack. The DDD has a laser gun that is able to eliminate Demogorgons, but the device takes a lot of time to charge. In general, charging the laser for j days will allow it to eliminate d_j Demogorgons.

Example: Suppose $(a_1, a_2, a_3, a_4) = (1, 10, 10, 1)$ and $(d_1, d_2, d_3, d_4) = (1, 2, 4, 8)$. The best solution is to fire the laser at times 3, 4 in order to eliminate 5 Demogorgons.

1. Construct an instance of the problem on which the following "greedy" algorithm returns the wrong answer:

   BADLASER$((a_1, a_2, a_3, \ldots, a_n), (d_1, d_2, d_3, , \ldots, d_n))$:
   Compute the smallest j such that $d_j \geq a_n$, Set $j = n$ if no such j exists
   Shoot the laser at time n
   if $n > j$ then BADLASER$((a_1, \ldots, a_{n-j}), (d1, \ldots, d_{n-j}))$

   Intuitively, the algorithm figures out how many days (j) are needed to kill all the Demogorgons in the last time slot. It shoots during that last time slot, and then accounts for the j days required to recharge for that last slot, and recursively considers the best solution for the smaller problem of size $n - j$.

   Solution:

   Let $(a_1, a_2, a_3, a_4) = (1, 1, 8, 2)$ and $(d_1, d_2, d_3, d_4) = (1, 2, 9, 1)$, then the greedy algorithm produces the answer that we should shoot on the first day, the second day, and the last day, and we can kill $1 + 1 + 2 = 4$ Demogrogons.

   However, if we shoot on the third and the last day, we can kill $8 + 1 = 9$ Demogorgons. Thus, the greedy algorithm returns the wrong answer, as $9 > 4$.

2. Given an array holding a_i and d_j, devise a dynamic programming algorithm that eliminates the maximum number of Demogorgons. Analyze the running time of your solution. *Hint: it is always optimal to fire during the last time slot.*

   Solution:

   We can apply a bottom-up dynamic programming structure to iteratively calculate and store the number of demogorgons that can be killed by charging up to k days and the last shooting on the last day, jth day. Let the function be $numKilled(MaxChargePeriod, LastShootDay)$. Let the memory dictionary be *memo*. Let the strategy dictionary be *strategy*. We start by calculating the base cases of $MaxChargePeriod = 1$ and iterating for all n $LastShootDay$, which has the result of d_1.

Then we have the result of $numKilled(MaxChargePeriod = 1, LastShootDay = k)$ for all $1 \leq k \leq n$. We store those values according to the memory dictionary $memo[1][k]$.

Then we continue to iterate through $MaxChargPeriod = l$ for each $LastShootDay = k, k \geq l$. Each of these larger cases with $MaxChargPeriod = l$, $LastShootDay = k$, as each larger case can be divided into summations of different combinations of smaller cases. For each $MaxChargPeriod = l$, $LastShootDay = k$, we can divide into $numKilled(l - i, k - i) + d_i$, for all $0 \leq i \leq k$. Then the result of $numKilled(MaxChargePeriod = l, LastShootDay = k)$ is $\max(numKilled(l - i, k - i) + d_i), 0 \leq i \leq k$.

We then update $memo[l][k] = numKilled(MaxChargePeriod = l, LastShootDay = k)$. We also want to keep the maximized strategy with charge days i for the shoot on kth day, so we update $strategy[k] = l$.

For example, let $MaxChargPeriod = 2$, $LastShootDay = k$. Then the larger case can be divided into two situations: 1. Charge for one day and shoot on $k - 1$th day, and then charge for another day and shoot again on kth day. 2. Charge for two days and only shoot on kth day. The first case has the result as the sum of known values in the *memo* dictionary, while the second case has the result of d_2. Then we compare results from all possible cases, two results in the example, and return the maximum of them as the value of $numKilled(MaxChargePeriod = 1, LastShootDay = k)$. Then we update $memo[2][k]$.

Thus, after calculating all cases, the final answer of the maximum number of Demogrogons that can be killed is $memo[n][n]$. The shooting days can be retrieved by recursively calling $shootingDay$.append(n), $n = n - strategy[n]$. Charging periods are the corresponding values of shooting days as keys in the strategy dictionary.

Finally, the time complexity is $O(n^3)$, as to iterate all cases, we need a time complexity of n^2, and under each case, there are up to n types of combination with another smaller case.